NASA-CR-201018

# Annual Progress Report "Tools for Analysis and Visualization of Large Time-Varying CFD Data Sets" NAS/NASA Ames Grant NAG 2-991 August 1, 1995 to May 15, 1996

Jane Wilhelms and Allen Van Gelder, Principal Investigators

# Parallel Hierarchical Direct Volume Rendering for Irregular and Multiple Grids

Jane Wilhelms, Allen Van Gelder, Paul Tarantino, Jonathan Gibbs
Computer Science Dept, Room 225AS, Santa Cruz CA 95064, USA.
E-mail: wilhelms, avg, pault, jono @cs.ucsc.edu
University of California, Santa Cruz 95064

**Abstract**

A general volume rendering technique is described that efficiently produces images of good quality from data defined over multiple, non-convex, irregular grids, which may also intersect each other. Such grids present problems for most current volume rendering techniques, which implicitly assume that cells do not intersect except on their boundaries. Also, the wide range of cell sizes (by a factor of 10,000 or more) presents difficulties for many techniques. This technique also permits inclusion of polygon mesh surfaces within the volume. Rendering is done in software, which eliminates the need for special graphics hardware, and also eliminates any artifacts associated with graphics hardware.

The volume is organized into a spatial hierarchy, making it possible to access data from a restricted region efficiently. The hierarchy has greater depth in regions of greater detail, as determined by the number of irregular cells in the region. This also makes it possible to render useful "preview" images very quickly by displaying each region associated with a tree node as one cell, a facility that is important for navigating in very large data sets.

The algorithmic techniques include use of a $k$-d tree, with prefix-order partitioning of triangles, to reduce the number of primitives that must be processed for one rendering, coarse-grain parallelism for a shared-memory MIMD architecture, a new perspective transformation that achieves greater numerical accuracy, and a scanline algorithm with depth sorting and a new clipping technique.

## 1 Introduction

Direct volume rendering is an attractive technique because it can convey a great deal of information in a single image by mapping the scalar data values in a sample volume to color and opacity. However, a great amount of computation is required to calculate the information presented, particularly when the samples are on irregular grids, which may even be intersecting multiple grids within the same 3D space. The problem is further exacerbated when the data sets are very large.

This paper presents a direct volume rendering technique developed for wide range of grids, including curvilinear grids with hexahedral cells, tetrahedral irregular grids, rectilinear grids, and intersecting multiple grids. The pertinent contributions are as follows:

1. A *software scan conversion* technique for direct volume rendering: The faces of volume grid cells are treated as independent polygons, much as in a scan conversion renderer for polygon rendering. However, to accommodate semi-transparency, they are fully sorted in screen depth at each pixel, and their values at each pixel are used in an integration in depth that simulates the accumulation of color and opacity between the faces. This technique is quite general because polygons are permitted to intersect. Thus, additional geometric elements, represented by polygons, can be included among the polygons from the volume, without any change to the basic technique. Because the process is entirely in software, it is portable, requiring no special graphics hardware.

2. *Multi-resolution hierarchy*: Polygons are associated with a $k$-d tree over the bounding box of the volume, and an approximation of the data values within the subregion is stored in each node of the tree. The use of a hierarchy has two advantages. First, it allows one to ignore large regions of the data that are invisible in a particular image. Second, it allows one to render either the detailed data or a faster, error-controlled approximation of it.

3. *Parallelization*: The method has been parallelized for shared-memory MIMD workstations. We observed a speed-up of 3.25 on four processors. However, as discussed in Section 3.2, the current implementation is not expected to scale up well beyond 16 processors.

## 2 Background and Related Work

Direct volume rendering originally was used for rectilinear grids, which permit many kinds of optimization. The two early approaches were *ray-casting* and *cell projection* [7]. *Splatting* (weighted voxel projection) soon followed [30]. Newer methods for regular grids include *hardware-assisted approaches* [16, 33], *Fourier volume rendering* [17, 21], further *shearing approaches* [15, 24], and *3D hardware textures* [3, 6, 12, 36].

However, many applications create volume data sets that are not rectilinear, such as computational fluid dynamics (CFD) and finite element analysis (FEM). Such data is often found on *curvilinear grids* (where a computational regular grid is warped to fit around objects of interest), and *unstructured grids* (where data points are connected to form tetrahedral or other polyhedral cells). Multiple overlapping and intersecting grids may be used to sample space around very complex shapes, such as the space shuttle [2]. Our research concentrates on rendering such irregular data.

A number of complexities are introduced when volume rendering irregular grids, which are not present with rectilinear grids. A visibility ordering (front-to-back) is not implicit. Many operations are much more expensive, such as intersecting rays with cells, projecting irregular cells, and interpolating across faces or through cells. Intersecting grids are a particular problem for most methods.

Ray-casting irregular grids tends to be quite slow, though parallelization can help significantly [8, 4, 26, 14, 19]. Faster projection methods have been developed in software [9, 10, 18, 23, 35] and in hardware [25, 33]. These hardware approaches trade some image quality for speed. Irregular grids can be resampled for speed [22, 31], also with loss in image quality.

Hierarchies have been shown to be helpful in accelerating direct volume rendering on regular grids [16, 34] and tetrahedral grids [5].

## 3 The Algorithm

This section first describes the basic scanline algorithm for volume rendering a region; then how it is parallelized; and finally how it is used with a multi-resolution hierarchy.

### 3.1 The Scanline Algorithm

The scanline technique has the merits of both generality and coherence. It is based on Watkin's scanline algorithm [29], with several modifications. Sample data values form the corners of polyhedral cells whose faces are (possibly non-planar) polygons. These faces are processed independently in the algorithm. In our implementation, we subdivide faces where necessary to form triangles; the algorithm could also be written to accommodate any polygon. Further details on this aspect of the algorithm can be found in a technical report [32].

While a great deal of work has been done (mostly in the 1980's) on the problem of scanline rendering of polygonal data sets, we wish to emphasize that using scan conversion for direct volume rendering is a different problem. Most importantly, in polygon scan conversion it is the *surfaces*, which are usually opaque, that have color properties. In direct volume rendering, it is the material *between* the cell surfaces that has color properties, and this material is often semi-transparent.

#### 3.1.1 Polygon Creation

The volume is decomposed into polygons, each given a sequential identifier (*Id*). Polygons consist of a list of pointers into a vertex list. Whenever a new geometrical transformation is applied, vertices are converted from world space to screen space by passing each vertex through the geometry and projection matrices. (See Section 3.5 for a description of the projection used, which is designed to preserve screen-depth distances near the center of the volume.) The polygon Id can be used as an index to find the locations of the vertices and their scalar data values. Because multiple grids are allowed, as well as surface polygon meshes, a grid Id specifies with which grid or surface this polygon is associated.

Any group of polygons to be rendered includes the faces of a *restrict box*, which by default is the bounding box around the volume, but which can be adjusted by the user. In some cases, it is established during traversal of the hierarchy (see Section 3.3). The six faces of the restrict box are treated much like other polygons during processing, but take on a special meaning during pixel processing (see Section 3.1.4).

#### 3.1.2 Y-Bucket Processing

Each scanline has associated with it a *y-bucket* list consisting of the polygon Ids for those polygons that first appear on that scanline. (We process scanlines bottom-to-top.) Polygons are excluded from y-buckets if they are outside the restrict box, or lie completely between adjacent scanlines, or lie completely between vertical lines through a pair of adjacent pixel centers.

After computing y-bucket lists, each scanline must be processed and drawn into the software frame buffer. An active list containing the polygon Id's of those polygons contributing to a scanline is maintained. Before processing the current scanline, the previous scanline must be updated by adding in new polygons from the current scanline and removing any that are no longer active. The active list is implemented as an array of structures, each containing various polygon information, and the scanline at which this polygon can be deleted from the active list. For polygons on the boundary of a grid, the vertices are stored in counter-clockwise order when viewed from outside the grid.

### 3.1.3 X-Bucket Processing

When processing a particular scanline, pertinent information for each active polygon is transferred to an *x-bucket* data structure, which is associated with the leftmost pixel in which the polygon will appear. A linked list of these structures is associated with each x-bucket in front-to-back visibility order.

As the scanline is processed, another active list is maintained and updated for each pixel. The active list contains the data value and the screen depth for each triangle that contributes to that pixel. As with the vertical active list (Section 3.1.2), polygons that become inactive at the current pixel are deleted, and polygons in the current x-bucket are inserted, except that now the list is maintained in sorted order. The surviving polygons of the previous pixel's active list must be updated with newly interpolated values of field data and screen depth, corresponding to one pixel of horizontal movement. New screen depths may require rearrangement of polygons among survivors, and new polygons must be merged in, maintaining screen-depth order.

### 3.1.4 Pixel Processing

The active list for a particular pixel is traversed to accumulate the color and opacity for that pixel. Polygons are processed front to back. In our implementation, data values (interpolated to that pixel) for each pair of adjacent polygons are averaged, and the average is used as the parameter of a transfer function that provides a color and opacity value. Taking into account the line-of-sight distance between the two polygons, a color and opacity contribution for that region is calculated. This contribution is composited into a software (floating point) frame buffer. The equations for this standard process are found elsewhere, e. g. [33].

There is a slight further complexity. Because six restricting polygons delimit the region to be rendered, there are two "restriction" polygons in any pixel list that should affect the image – the first indicates contributions should begin to be accumulated, and the second indicates contributions should stop being accumulated. (This removes the need to explicitly clip any polygon.) If the first normal (non-restriction) polygon in the pixel's active list is not an *exterior face* of a grid, this signifies that rendering begins inside a grid, and there is "material" between the restriction polygon and the first normal polygon. The color and opacity of this material must be calculated by extrapolation, since the restriction polygon has no such properties. The same applies if the last normal polygon is not an *exterior face*.

Section 3.3 describes how to accurately draw a collection of smaller regions associated with traversing a hierarchy.

## 3.2 Parallelizing the Algorithm

Three primary components of this algorithm can, by and large, be easily parallelized. The first is the transformation of the vertices from world space to pixel space, and is trivially parallelizable, as each vertex can be transformed independently.

The second parallelizable component is the task of grouping polygons by scanline into y-buckets. Its parallelization is more involved, but highly scalable. It proceeds in two passes.

In the first pass, each processor is given an equal number of polygons to process. One temporary array stores, for each polygon, the lowest scanline where it appears, or an invalidation flag if it doesn't cross any pixel centers within the restrict box (as described in Section 3.1.2). A second, two-dimensional, array stores, for each scanline and processor, how many polygons the processor found that belong in that y-bucket.

After the first pass, the number polygons belonging in each y-bucket is found by accessing the latter array, and space for each y-bucket is allocated by partitioning a common output array. The space for one y-bucket is further partitioned into space for each processor to fill within that y-bucket. (While this transitional step can also be parallelized by standard techniques, its work per processor is only proportional to the number of scanlines, not the number of polygons.)

During the second pass, each process creates appropriate y-bucket structures for all polygons that it processed in the first pass, except those that were invalidated.

The third parallelizable component involves processing each scanline in order from bottom to top, which involves the bulk of the computation. There are several ways of implementing this part in parallel. Our implementation contains a "critical section" of code, in which a processor updates the current active list for the scanline, takes a copy of it, and then exits the critical section. Then it builds the x-buckets and processes the scanline, as described in Section 3.1.3. The next available processor waits to enter the critical section and gets the next available scanline.

This implementation is not 100% scalable since it contains a critical section that can act as a bottleneck as more processors are added. However, it was fairly easy to implement and caused low contention with the four processors we had available. Our measurements suggest that it can extend to about 16 processors before contention becomes a serious drawback.

Uselton described scanline processing as "embarrassingly parallel" for ray-casting [26]. However, the expensive part of ray-casting in irregular structures with semi-transparent material is the calculation of exit faces from each cell entered by the ray. (Moreover, this technique does not directly apply to intersecting grids, where a ray can be in two cells simultaneously.) The technique presented here uses coherence to greatly reduce the cost that corresponds to exit-face calculations, at the expense of introducing a short serial critical section.

### 3.3 Use with a Multi-Resolution Hierarchy

A method of spatial partitioning in $k$ dimensions, called $k$-d trees, was introduced by Jon Bentley [1]. A binary tree is built that splits in one dimension at a time. Our current implementation splits in a round-robin fashion, but one could easily adopt a more sophisticated policy based on the locations of objects. At tree node $v$, the (hyper-)plane that splits the region is orthogonal to the $x_i$-axis when splitting on dimension $i$. Our implementation bisects the region, but in general, any partitioning value $X_v$ can be chosen. Thus the hyper-plane is defined by $x_i = X_v$. Because of this orthogonality, $k$-d trees are distinct from binary space partitioning (BSP) trees. We have added a new adaptation to $k$-d trees, as described below.

#### 3.3.1 Creation of the Hierarchy

Each node $v$ in the tree has associated with it a list of polygon Ids. It is built as follows: for each polygon $p$ passed into the tree node, which is splitting on dimension $i$, if all vertices have an $x_i$ location less than

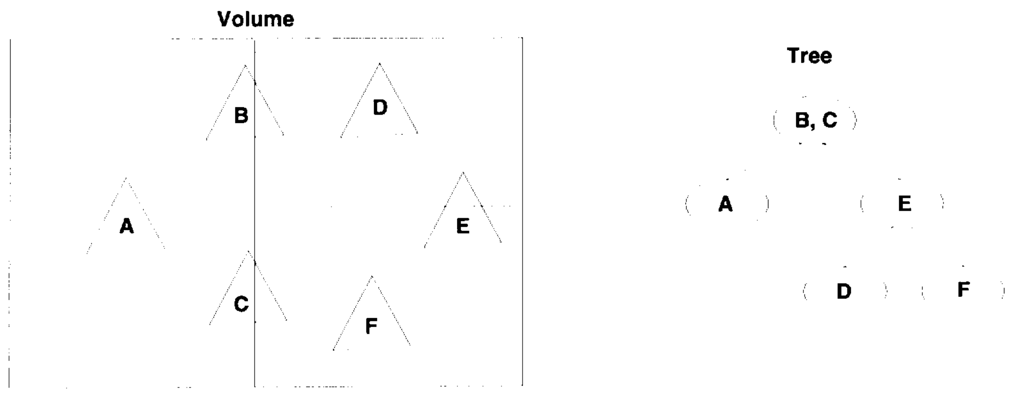

Figure 1: Distributing polygons in a $k$-d tree, as described in Section 3.3.

$X_v$, place $p$ in the set to be passed to the left subtree. If all vertices have an $x_i$ location that is greater than the bisection value, place $p$ in the set to be passed to the right subtree. Otherwise, retain $p$ in the set to be stored at $v$ (see Figure 1). Then, process the left and right subtrees. If the number of polygons passed into node $v$ is below a user-defined threshold, no splitting occurs, and the node is a leaf. Note this results in polygons being associated with the smallest node region that completely contains them, following Greene [11], and avoids subdividing polygons.

We developed a new, flexible storage method that allows us to avoid the extra space of linked lists and still pass all objects associated with a subtree in one operation. Polygon Ids are stored in one common array *Tid*, and arranged in the order that they would appear during a prefix-order traversal of the $k$-d tree. (A post-fix order can also be easily made to work.) Thus, all polygons stored in a subtree form a contiguous section in *Tid*, and all polygons associated with that subtree's root node are at the beginning of that segment.

Assuming the array begins with a known subscript of 0, it is necessary to store globally the total number of polygons in the tree, and to store in each node the offset to the *beginning* of its set of polygons. The remaining values can be recovered during traversal, provided the end-point for each subtree is passed in as a parameter of the traversal. For the whole tree, this is just the total number of polygons. Thus if a tree node $v$ is visited with end-point $E$ passed in:

- The end-point for polygons associated with node $v$ is the offset (begin-point) of $v$'s left child;
- The end-point for the left child is the offset of $v$'s right child;
- The end-point for the right child is $E$.

In this convention, an end-point is the first index *above* the segment.

### 3.3.2 Fully Detailed Rendering

To prepare for rendering, the tree is traversed and a set of subranges of polygon Ids is developed, which represent all polygons within the user-defined restrict box. Recall that all polygon Ids within the spatial region represented by one node are stored contiguously in the polygon array, *Tid* (Section 3.3.1). If the entire node is outside the restrict box, the traversal returns from that branch immediately.

If the node is entirely inside the restrict box, then the subrange for that node is added to the set of subranges to be rendered, and again the traversal returns without exploring the subtrees. The same applies for leaf nodes, and for nodes whose region contains fewer polygons than a user-specified cut-off.

If a node is partially inside the restrict box and none of the above exceptions applies, then the subrange of *Tid* associated with the node itself (but not its descendants) is added to the set of subranges to be rendered, and traversal continues into the children of this node, which are treated recursively in the same manner.

When this traversal is completed, we have a list (represented as a set of start and end positions within the polygon array *Tid*) of all possibly relevant polygons, which are then sent to the renderer for processing (see Section 3.1).

## 3.4 Multi-Resolution Rendering

For *multi-resolution rendering*, we wish to display error-controlled approximations of the data for speed. For this, each node must contain a model representing an approximation of the data in the subregion it represents, as well as an error term representing the deviation of the model from the data. Our multi-resolution technique departs from that associated with wavelets [20] in that each level represents a complete model of its region, whereas wavelets represent just the detail information not represented at higher levels.

For rendering, the user specifies an acceptable error. During the traversal, if the error associated with the node is less than the acceptable error, traversal stops there and the nodal region is drawn, rather than the individual polygons. To draw a nodal region, it is simply treated as the six bounding polygons defining it. The values at the eight corners of the region come from the data model.

At present, our nodal model is a rather trival one, but it constructs very fast "preview" images to help orient the user in a very large data set (see Figure 10). Eight data values are stored with each node representing the values of the eight node cell corners. These values are found, for each corner, by taking the average of any data values within the node that are nearest that corner. The error metric is the mean squared error between the average value over the entire node region and the actual vertex values in the region. We will next implement a trilinear model, as we used in [34]. We believe that finding a better data model for approximating and rendering is an important and interesting problem that we will pursue in future.

There are some differences in the rendering process caused by the fact that some regions may be drawn exactly, using grid polygons, while others are drawn approximately, using the nodal model. Rather than storing all possibly visible polygons, regions corresponding to each tree node at which traversal stops are composited individually into the software frame buffer.

Traversal occurs in front-to-back visibility order. When rendering a subregion, the algorithm described in Section 3.1 is used, with the local restrict box being the intersection of the boundary of the subregion and the global restrict box. As in fully detailed rendering, during traversal, the boundaries of the subregion represented by the node are tested and if the whole region lies outside the the restrict box, that subtree is not processed further.

When independently rendering a collection of small regions associated with a tree traversal, where some

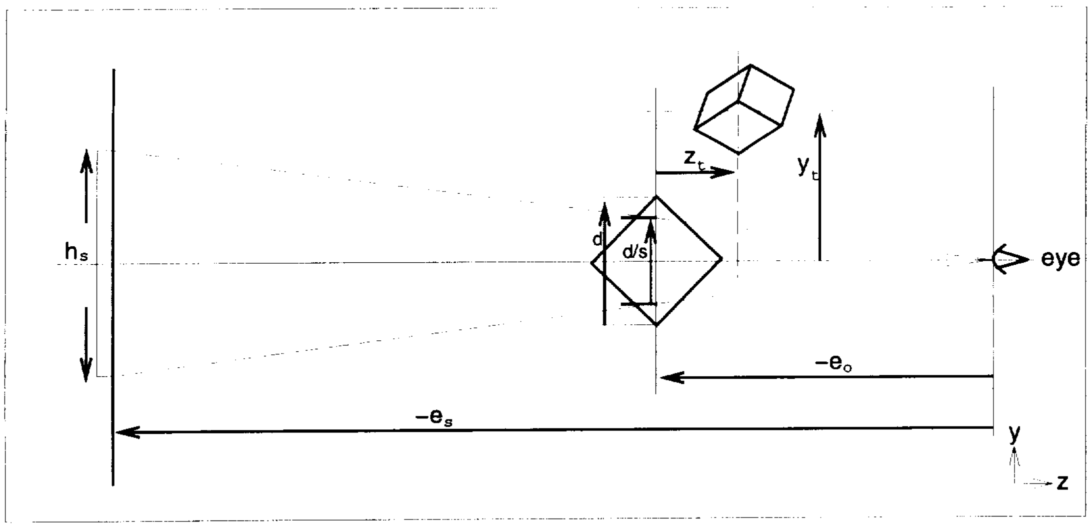

Figure 2: Figure of perspective view - eye is at (0,0,0) world space. Volume is projected onto screen at left.

regions may be rendered in full detail, special care must be taken to insure that drawing between the last polygon of the front region and the first of the next is done accurately. In addition, the last region that is rendered for a given pixel must be drawn all the way to the region border, provided that the region border lies inside the volume. Three screen buffers are maintained to achieve this accuracy. A data buffer keeps the data value of the last polygon that was drawn inside the region, and is used when drawing starts for the next region. A depth buffer is kept to represent the screen depth from the last polygon to the border of the region, and is used to calculate the distance to the first polygon inside the next region. Since the renderer does not know which region is the last to be drawn for a given pixel, a second data buffer is maintained that represents the data value at the region back border and is obtained either by using the value of the first polygon behind the region border, if one exists, or it is given the value of the other data buffer. After all the regions have been drawn, all three buffers are used to finish drawing any pixels that are not opaque yet. The pixels are drawn from the front data value to the back data value using the depth from the depth buffer.

### 3.5 Numerically Stable Perspective Transformation

A standard perspective transformation matrix transforms viewable world space into a unit viewing volume, so that the $Z$ values of the vertices are transformed to values close to each other, introducing substantial floating point *relative* errors and some ordering inconsistencies. We corrected this problem using a different transformation, described here. This transformation preserves accuracy of the $Z$ values near the volume center, by transforming the center to zero in screen $Z$. Symbols used in this section are summarized in Figure 3. The main innovation is the introduction of a translation in screen-Z, given by Eq. 2, which produces a numerically stable equation (Eq. 3) for screen-Z in terms of world-Z, near the world-Z center.

The volume being viewed is inside a user-definable visibility bounding box (Vbb), of diagonal $d$ (see Figure 2). The Vbb is first centered at (0,0,0) in world space, then it is rotated by the user rotations and translated back by $e_o$. Then the user translates (exaggerated in the diagram) are applied in screen space,

| Symbol | Definition | Units |
|---|---|---|
| $h_s$ | screen height | pixels |
| $e_s$ | distance from eye to screen | pixels |
| $e_o$ | distance from eye to volume origin | world units |
| $d$ | diagonal of volume's visible bounding box | world units |
| $s$ | user's scale factor | pure number |
| $z_t$ | translate in screen z | world units |
| $y_t$ | translate in screen y | world units |
| $x_t$ | translate in screen x | world units |
| $z_v$ | world vertex location z | world units |
| $y_v$ | world vertex location y | world units |
| $x_v$ | world vertex location x | world units |
| $z_s$ | z transformed to screen | pixels |
| $y_s$ | y transformed to screen | pixels |
| $x_s$ | x transformed to screen | pixels |
| $z_{so}$ | offset (defined in Eq. 2) | pixels |

Figure 3: Symbols used in perspective calculation.

with the restriction that

$$z_t + \frac{1}{2}d + NearClip \leq e_o$$

so that the denominator ($e_o - z_t - z_v$) is positive. (Our implementation uses $NearClip = .05e_o$.),

Using similar triangles, we get the following relationships

$$\begin{aligned} e_s &= \frac{s \cdot h_s \cdot e_o}{d}, \\ \frac{x_s}{e_s} &= \frac{x_v + x_t}{e_o - z_t - z_v}, \\ \frac{y_s}{e_s} &= \frac{y_v + y_t}{e_o - z_t - z_v}, \\ \frac{z_s + z_{so}}{e_s} &= \frac{z_v + z_t}{e_o - z_t - z_v} = 1 - \frac{e_o}{e_o - z_t - z_v}. \end{aligned}$$

Conversion to screen space is

$$\begin{aligned} x_s &= \frac{e_s(x_v + x_t)}{e_o - z_t - z_v} \\ y_s &= \frac{e_s(y_v + y_t)}{e_o - z_t - z_v} \\ z_s &= \frac{e_s(z_v + z_t)}{e_o - z_t - z_v} - z_{so} \end{aligned} \qquad (1)$$

where the quantity $z_{so}$ is an offset to the screen position that is chosen so that the center of the volume is mapped to 0 in $z_s$.

(The center of the volume might map anywhere in screen X and Y, as it is not necessarily within the

Vbb.) The required value is

$$z_{so} = \frac{e_s \cdot z_t}{e_o - z_t} \tag{2}$$

which gives the numerically stable expression

$$z_s = \frac{e_s \cdot z_v \cdot e_o}{(e_o - z_t)(e_o - z_t - z_v)} \tag{3}$$

To invert the $z_v$-to-$z_s$ mapping, just solve for $z_v$. The following equation gives the difference in world-Z between two points along one sight line ($z_{v2} - z_{v1}$), in terms of their screen-space values, again in a numerically stable form:

$$z_{v2} - z_{v1} = \frac{e_o \cdot e_s(z_{s2} - z_{s1})}{\left(\frac{e_o \cdot e_s}{e_o - z_t} + z_{s2}\right)\left(\frac{e_o \cdot e_s}{e_o - z_t} + z_{s1}\right)} \tag{4}$$

Finally, the world *distance* between two points on a sight line is inversely proportional to the cosine of the angle $\theta$ between the sight line and the screen-Z axis:

$$dist = \frac{z_{v2} - z_{v1}}{cos(\theta)} = (z_{v2} - z_{v1})\sqrt{\frac{z_s^2}{x_s^2 + y_s^2 + z_s^2}} \tag{5}$$

Accurate values of this distance are critical because they affect the compositing of color through possibly thousands of triangles that are very close to each other, as seen in the space-shuttle grid (see Figure 9).

## 4 Experimental Results

In our experiments, we studied the performance of the new renderer. We first compare the performance of the method on a curvilinear multi-grid using single and multiple processors. Next, we compare the performance of the algorithm against other renderers we have implemented. Finally, we explore the ramifications of the hierarchy.

For our results, we used the following data volumes: the *blunt fin* [13] (a single curvilinear volume of 40,960 data points); the *space shuttle* [2] (a nine-grid curvilinear volume consisting of 941,159 data points); the *lftrx* (*fighter jet*, courtesy of John Batina of NASA Langley Research Center (an unstructured tetrahedral grid with accompanying polygon surface file consisting of 13,832 data points and 70,125 tetrahedra); and, for comparison to rectilinear grid renderers, the *hipip* molecular data set courtesy of L. Noodleman and D. Case, Scripps Clinic (a 64x64x64 resolution rectilinear volume, or 262,144 data points); and the rectilinear *CTHead* data set from UNC (at a resolution of 200x200x50 or 2,000,000 sample points).

### 4.1 Performance on a Single Processor and in Parallel

For this evaluation, we used the nine-grid space shuttle data set. (Figure 9 shows four volume renderings of this data set, with the shuttle itself treated as a polygonal surface.) Results are presented in graphical form with an emphasis on showing the gains in speed for one to four processors. The dataset was rendered with a spatial rotation of ($-90°X$, $-10°Y$, $0°Z$), and scale factors that ranged from 1.0 to 5.16. A scale factor of

| Scale - | 1.00 | 1.20 | 1.44 | 1.73 | 2.07 | 2.49 | 2.99 | 3.58 | 4.30 | 5.16 |
|---|---|---|---|---|---|---|---|---|---|---|
| Active Polygons (Million) | 0.865 | 0.995 | 1.153 | 1.346 | 1.495 | 1.660 | 1.836 | 2.022 | 2.192 | 2.360 |
| Single processor | 54.39 | 66.87 | 83.52 | 98.88 | 112.55 | 125.34 | 138.02 | 149.15 | 161.75 | 173.80 |
| Two processors | 30.66 | 37.46 | 46.34 | 56.10 | 63.30 | 71.32 | 77.67 | 84.33 | 90.17 | 97.57 |
| Three processors | 21.30 | 26.19 | 32.58 | 38.82 | 44.10 | 49.36 | 54.24 | 58.26 | 62.70 | 68.15 |
| Four processors | 16.85 | 20.16 | 25.06 | 30.10 | 34.14 | 37.89 | 41.62 | 44.89 | 49.15 | 53.03 |

Table 1: Elapsed time comparisons (in seconds) on an SGI Onyx using four 150-MHz processors, with 256MB memory. (NASA space shuttle data set)

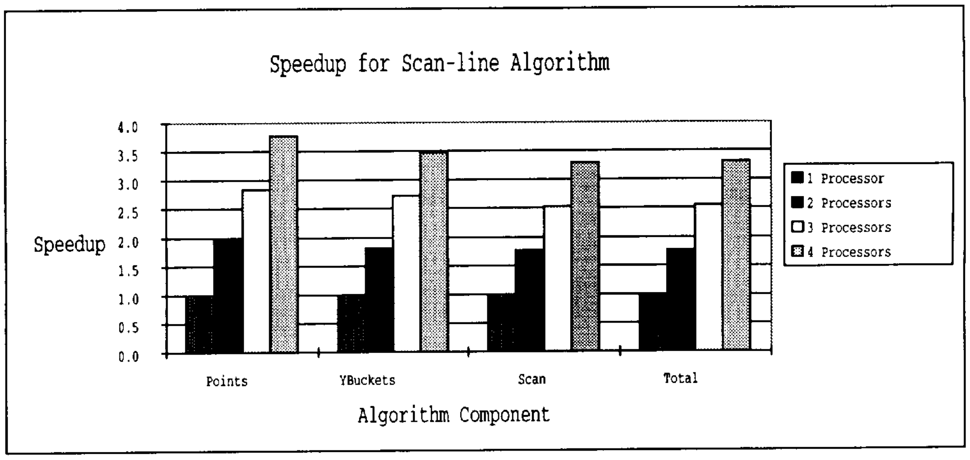

Figure 4: Speedup of algorithm for shuttle (scale = 3.58).

1.0 is the default situation where the long diagonal of the bounding box of the volume is equal to the width and height of the screen window. The default window resolution is 500x500 pixels.

The speedup data for four processors in particular is not exact since the machine, an SGI Onyx, had four 150-MHz processors, which means that some time was spent by one or more processors on unrelated processes. This introduced some variation in performance, but trends in speed can still be observed for four processors.

### 4.1.1 Timing analysis

We looked separately at the cost and parallelizability of different sections of the algorithm, including transformation of the points to screen space, creation of the Y-buckets, and generating scanlines. Speed-up factors for these sections using parallel processors is shown in Figure 4.

The average total CPU time to transform the shuttle points from world space to screen space was 1.25

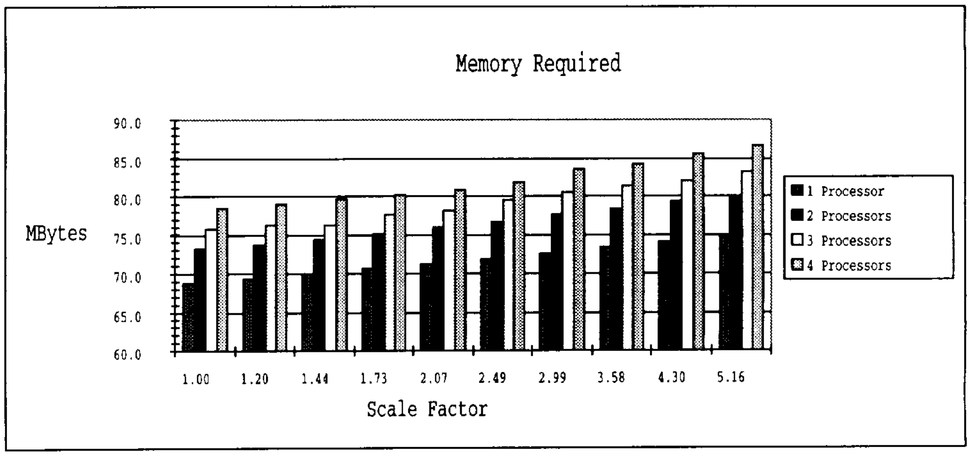

Figure 5: Amount of memory used by the program.

seconds. Figure 4 shows the speedups for transforming the points to screen space for multiple processors. shows that this part of the algorithm is scalable with a greater than 3.5 speedup for four processors.

CPU times for creating Y-buckets varied from 16.8 seconds to 18.9 seconds. The time needed to create Y-buckets increases slightly (less than 10% change from scale = 1.0 to scale = 5.16). This increase in time is due to the fact that the Y-bucket array grows in size and active triangle id's must be stored in the array. An active triangle is one that is not clipped from the drawing area due to size or position on the screen.

The speedup factors for generating scanlines from Y-buckets are not as high as those for transforming points to screen space or building Y-buckets because of the critical code mentioned in Section 3.2. However, the increase in speed is still significant for four processors.

The total elapsed times to render each picture are presented in Table 1, with overall speedup factors are in Figure 4. It is clear that although 100% scalability is not achieved, there is a significant speedup (approximately 3.25) for four processors.

### 4.1.2 Memory

Memory use is an important factor in measuring the efficiency of an algorithm. Figure 5 shows the memory (in Megabytes) used by the program for a range of scale factors and multiple processors. This information was recorded by keeping statistical information associated with all calls to *malloc()* and *free()*. Most of the memory was used for storage of the following major items:

- Space shuttle grid - 33.9 Mbytes.
- Transformed points - 11.3 Mbytes.
- Frame buffer (R,G,B,Alpha,Composite) - 5.0 Mbytes
- Y-bucket array - 11.0 Mbytes

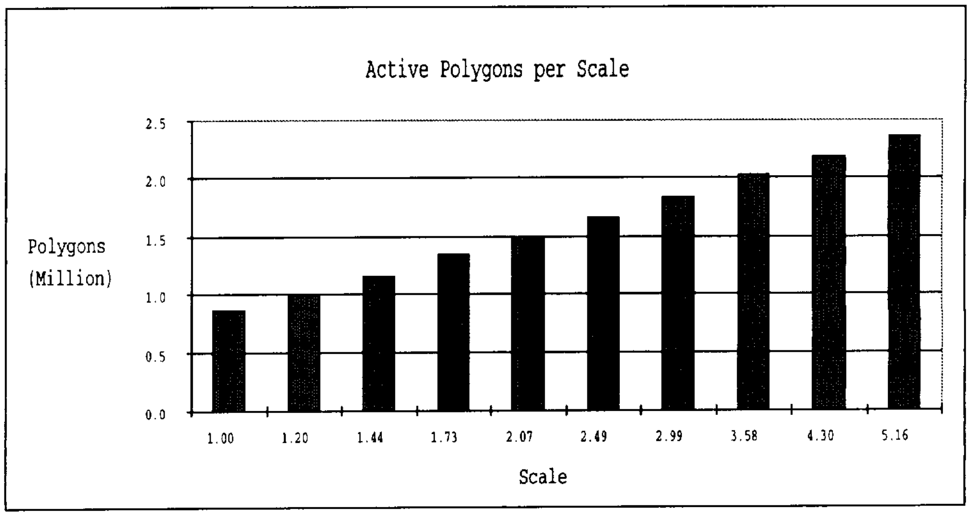

Figure 6: Number of active polygons for different scale factors.

The rest of the memory is used by the program to keep track of the triangles that it is rendering. It can be seen from Figure 5 that memory usage increases with the number of processors. This is due to the fact that each processor is rendering a scanline and needs the memory to hold the data structures for that scanline, such as the active polygon list and the X-buckets. Also, as the number of active polygons increase, the memory needed by active lists and buckets increases as well.

### 4.1.3 Other factors

It is clear from Figures 6 and 7 that as the window size is increased or as the scale is increased, the number of active polygons is increased as well. As the scale is increased, there is an upper limit to the number of active polygons which is less than the total number of polygons in the data set because at high scale factors, more of the volume is projected off of the screen and the polygons that project off of the screen are discarded by the algorithm during Y-bucket creation. Increasing the window size will allow very small polygons, that may have been discarded because they did not cross a pixel, to become active since they are now projected onto a larger portion of the screen.

## 4.2 Comparison to Other Renderers

We have done some speed comparisons of this new renderer against other direct volume renderers we have written. None of these other renderers have the generality to handle irregular multi-grids like the space-shuttle (which most renderers will not handle), and we have no other renderer that handles tetrahedral volumes. However, we can compare image quality and performance against the following:

1. *Ray Casting* for rectilinear grids [27].

2. *Coherent Projection* (hardware Gouraud shading) for rectilinear grids [33].

3. *Incoherent Projection* (hardware Gouraud shading) for single curvilinear grids [28].

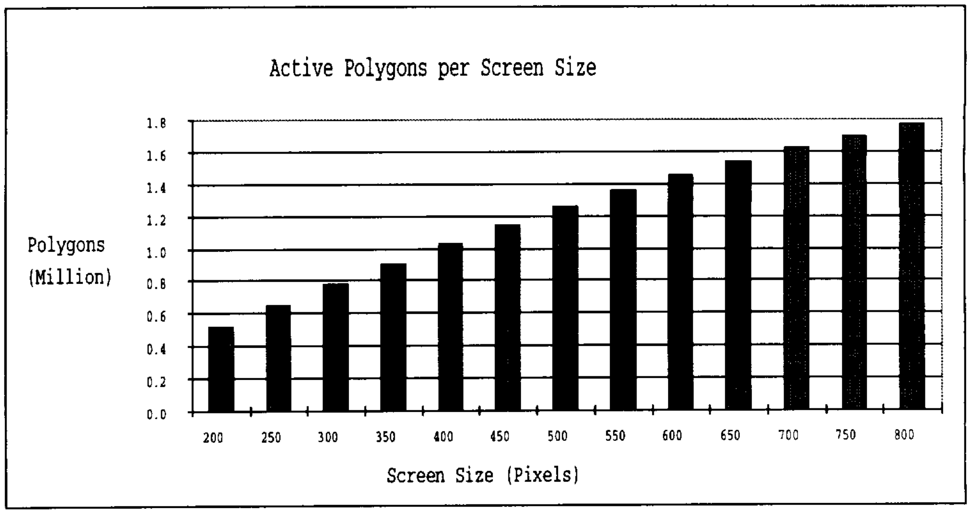

Figure 7: Number of active polygons for different screen sizes.

Note that because of its extreme generality, our new software scan conversion algorithm cannot compete with methods designed to take advantage of the simplicity of regular grids.

Table 2 compares the time taken to render each of the data sets described above by the renderers capable of handling them. Each volume is drawn in a 500x500 pixel window at a scale of 1 or 5 times.

We do not have space to show complete comparisons of image quality on all these methods. Figure 8 does compare the four methods on the hipip data set, which is rectilinear and thus amenable to all methods. It should be mentioned that the differences between the methods are brought out in this image by using a transfer function that abruptly changes from color to transparent. If a more smoothly varying function is used, the differences between methods can be very minor. We can provide some further general comments. First, software scan conversion generally produces pictures equivalent in image quality to ray casting much faster, because of the use of coherence. However, our particular projection methods are not optimized for opaque data sets, while a ray caster processing front to back can easily halt when the pixel becomes opaque. Thus on a data set such as the head, the ray tracer can be competitive or even faster than other methods. There are methods of including early termination due to opacity in projection methods; we haven't implemented them yet.

In general, software scan conversion produces a noticeably clearer image with less artifacts than the hardware Gouraud shading methods. While it is noticeably slower, this cost may often we worth it for the improvement in image quality.

Figure 9 shows four images of the space shuttle at different scales.

### 4.3 Performance with the Hierarchy

There are two aspects of the hierarchy. First, there is the temporal savings of discarding whole invisible subregions at one time, rather than examining their primitives individually. We found this a significant savings only when zoomed in on the volume considerably, though with large volumes, the gains may be more

| Rendering Method - | Software Scan Conversion | Incoherent Projection | Coherent Projection | Ray Casting |
|---|---|---|---|---|
| Space Shuttle scale 1 | 60.9 | | | |
| Space Shuttle scale 5 | 202.1 | | | |
| Blunt Fin scale 1 | 19.4 | 16.4 | | |
| Blunt Fin scale 5 | 51.4 | 16.5 | | |
| Fighter Jet scale 1 | 24.6 | | | |
| Fighter Jet scale 5 | 92.6 | | | |
| Hipip scale 1 | 74.5 | 109.4 | 12.2 | 63.0 |
| Hipip scale 5 | 128.9 | 109.4 | 1.8 | 187.0 |
| CTHead scale 1 | 180.1 | | 56.7 | 48.0 |
| CTHead scale 5 | 203.9 | | 9.46 | 140.0 |

Table 2: Speed Comparison of Various Renderers (C.P.U. seconds) on an SGI Onyx with Reality Engine II graphics, using one 150-MHz processor. Only renderers relevant to a particular data type are shown. Projection methods utilize the Reality Engine graphics, while ray-casting and scan conversion do not.

| Scale Factor | 1 | 4 | 16 | 64 | 256 |
|---|---|---|---|---|---|
| Speedup With Hierarchy | 0.95 | 0.98 | 1.15 | 1.37 | 1.96 |

Table 3: Speedup gained on space shuttle using a hierarchy to avoid invisible regions, on an SGI Onyx using one 150-MHz processor.

significant. This can be seen in Table 3, which shows the speedup gained by using the hierarchy at different levels of zoom.

The second aspect is the use of a multi-resolution model at some nodes to approximate the information. As we explained above, our explorations in this area for irregular grids is just begun. Figure 10 shows a comparison of the fighter jet shown rendering each primitives and using an error approximation. The left column shows the volume and embedded surface, the center column the volume only, and the right column the approximation of the volume. The top row shows the whole data set, the bottom is zoomed in by a factor of two on an interested area. The unapproximated images took between 38 and 44 seconds to render, while the two approximated images took less than 1 second.

## 5 Conclusions

The renderer described in this paper allows rendering of large multiple intersection irregular and regular grids including polygonal meshes without the use of expensive graphics hardware. Factors such as screen size and scale can affect the time needed to render the volume. The renderer is parallelizable and measurements show that this can greatly reduce elapsed time without greatly increasing memory requirements. Use of a k-d tree makes the algorithm better able to handle very large dta sets. Accurate depth calculation can be achieved by using the project method described in Section 3.5.

## Acknowledgements

Funds for the support of this study have been allocated by NAS/NASA-Ames Research Center, Moffett Field, California, Grant Number NAG-2-991, and by the National Science Foundation, Grants Number CCR-9503829 and CDA-9115268.

## References

[1] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):214–229, 1975.

[2] P.G. Buning, I.T. Chiu, Jr. F.W. Martin, R.L. Meakin, S. Obayashi, Y.M. Rizk, J.L. Steger, and M. Yarrow. Flowfield simulation of the space shuttle vehicle in ascent. *Fourth International Conference on Supercomputing*, 2:20–28, 1989. Space Shuttle data reference.

[3] Brian Cabral, Nancy Cam, and Jim Foran. Accelerated volume rendering and tomographic reconstruction using texture mapping hardware. In *1994 Symposium on Volume Visualization*, pages 91–98, Washington, D.C., October 1994.

[4] Judy Challinger. Parallel volume rendering for curvilinear volumes. In *Proceedings of the Scalable High Performance Computing Conference*, pages 14–21. IEEE Computer Society Press, April 1992.

[5] Paolo Cignoni, Leila De Floriani, Claudio Montani, Enrico Puppo, and Roberto Scopigno. Multiresolution modeling and visualization of volume data based on simplicial complexes. In Arie Kaufman and Wolfgang Krueger, editors, *1994 Symposium on Volume Visualization*, Washington, D.C., October 1994. ACM.

[6] T. J. Cullip and U. Newman. Accelerating volume reconstruction with 3d texture hardware. Technical Report TR93-027, University of North Carolina, Chapel Hill, N. C., 1993.

[7] James D. Foley, Andies Van Dam, Steven Feiner, and John Hughes. *Computer Graphics: Principles and Practice.* Addison-Wesley Publishing Company, Reading, Mass., 2 edition, 1990.

[8] Michael P. Garrity. Raytracing irregular volume data. *Computer Graphics*, 24(5):35-40, December 1990.

[9] Christopher Giertsen. Volume visualization of sparse irregular meshes. *IEEE Computer Graphics and Applications*, 12(2):40-48, March 1992.

[10] Christopher Giertsen and Johnny Peterson. Parallel volume rendering on a network of workstations. *IEEE Computer Graphics and Applications*, pages 16–23, November 1993.

[11] Ned Greene, Michael Kass, and Gavin Miller. Hierarchical z-buffer visibility. *Computer Graphics (ACM SIGGRAPH Proceedings)*, 27:231–238, August 1993.

[12] S. Guan and R. G. Lipes. Innovative volume rendering using 3d texture mapping. In *SPIE: Medical Imaging 1994: Images Captures, Formatting and Display.* SPIE 2164, 1994.

[13] Ching-Mao Hung and Pieter G. Buning. Simulation of blunt-fin-induced shock-wave and turbulent boundary-layer interaction. *J. Fluid Mechanics*, 154:163–185, 1985.

[14] Koji Koyamada. Fast traversal of irregular volumes. In T. L. Kunii, editor, *Visual Computing - Integrating Computer Graphics and Computer Vision*, pages 295–312. Springer Verlag, 1992.

[15] Philippe Lacroute and Marc Levoy. Fast volume rendering using a shear-warp factorization of the viewing transformation. *Computer Graphics (ACM Siggraph Proceedings)*, pages 451–458, July 1994.

[16] David Laur and Pat Hanrahan. Hierarchical splatting: A progressive refinement algorithm for volume rendering. *Computer Graphics (ACM Siggraph Proceedings)*, 25(4):285–288, July 1991.

[17] Marc Levoy. Volume rendering using the fourier projection-slice theorem. In *Proceedings of Graphics Interface '92*, Vancouver, B.C., 1992. Also Stanford University Technical Report CSL-TR-92-521.

[18] Bruce Lucas. A scientific visualization renderer. In *Visualization '92*, pages 227–233. IEEE, October 1992.

[19] Kwan-Liu Ma. Parallel volume ray-casting for unstructured grid data on distributed memory architectures. In *1995 Parallel Rendering Symposium*, pages 23–30. ACM, 1995.

[20] S. G. Mallat. A theory for multiresolution signal decomposition: The wavelet representation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 11(7):674–693, 1989.

[21] Tom Malzbender. Fourier volume rendering. *ACM Transactions on Graphics*, 12(3):233–250, July 1993.

[22] Xiaoyang Mao, Lichan Hong, and A. Kaufman. Splatting of curvilinear volumes. In *Visualization '95*, pages 61–68, San Jose, CA, November 1995. IEEE.

[23] Nelson Max, Pat Hanrahan, and Roger Crawfis. Area and volume coherence for efficient visualization of 3d scalar functions. *Computer Graphics (ACM Workshop on Volume Visualization)*, 24(5):27–33, December 1990.

[24] Peter Schroeder and James B. Salem. Fast rotation of volume data on data parallel architectures. In *Visualization '91*, pages 50–57. IEEE, 1991.

[25] Peter Shirley and Allan Tuchman. A polygonal approximation to direct scalar volume rendering. *Computer Graphics*, 24(5):63–70, December 1990.

[26] Sam Uselton. Parallelizing volvis for multiprocessor sgi workstations. Technical Report RNR-93-013, NAS-NASA Ames Research Center, Moffett Field, CA, 1993.

[27] Allen Van Gelder, Kwansik Kim, and Jane Wilhelms. Hierarchically accelerated ray casting for volume rendering with controlled error. Technical Report UCSC-CRL-95-31, University of California, Santa Cruz 95064, Santa Cruz, CA 95064, March 1995.

[28] Allen Van Gelder and Jane Wilhelms. Rapid exploration of curvilinear grids using direct volume rendering. Technical Report UCSC-CRL-93-02, University of California, Santa Cruz, UCSC CIS Department, Applied Sciences Building, Santa Cruz, CA 95064, 1993. (extended abstract in *Proc. IEEE Visualization 93*, Oct. 1993).

[29] G.S. Watkins. *A Real Time Visible Surface Algorithm*. PhD thesis, University of Utah, Salt Lake City, June 1970.

[30] Lee Westover. Footprint evaluation for volume rendering. *Computer Graphics (ACM Siggraph Proceedings)*, 24(4):367–76, August 1990.

[31] Jane Wilhelms, Judy Challinger, Naim Alper, Shankar Ramamoorthy, and Arsi Vaziri. Direct volume rendering of curvilinear volumes. *Computer Graphics*, 24(5):41–47, December 1990. Special Issue on San Diego Workshop on Volume Visualization.

[32] Jane Wilhelms, Paul Tarantino, and Allen Van Gelder. A scan-line algorithm for volume rendering of multiple curvilinear grids. Technical Report UCSC-CRL-95-57, University of California, Santa Cruz, Santa Cruz, Ca 95064, 1995.

[33] Jane Wilhelms and Allen Van Gelder. A coherent projection approach for direct volume rendering. *Computer Graphics (ACM Siggraph Proceedings)*, 25(4):275–284, 1991.

[34] Jane Wilhelms and Allen Van Gelder. Multi-dimensional trees for controlled volume rendering and compression. In *ACM Workshop on Volume Visualization 1994*, Washington, D.C., October 1994. See also technical report UCSC-CRL-94-02.

[35] Peter Williams. Interactive splatting of nonrectilinear volumes. In *Visualization '92*, pages 37–44. IEEE, October 1992.

[36] Orion Wilson, Allen Van Gelder, and Jane Wilhelms. Direct volume rendering via 3d textures. Technical Report UCSC-CRL-94-19, CIS Board, University of California, Santa Cruz, 1994.

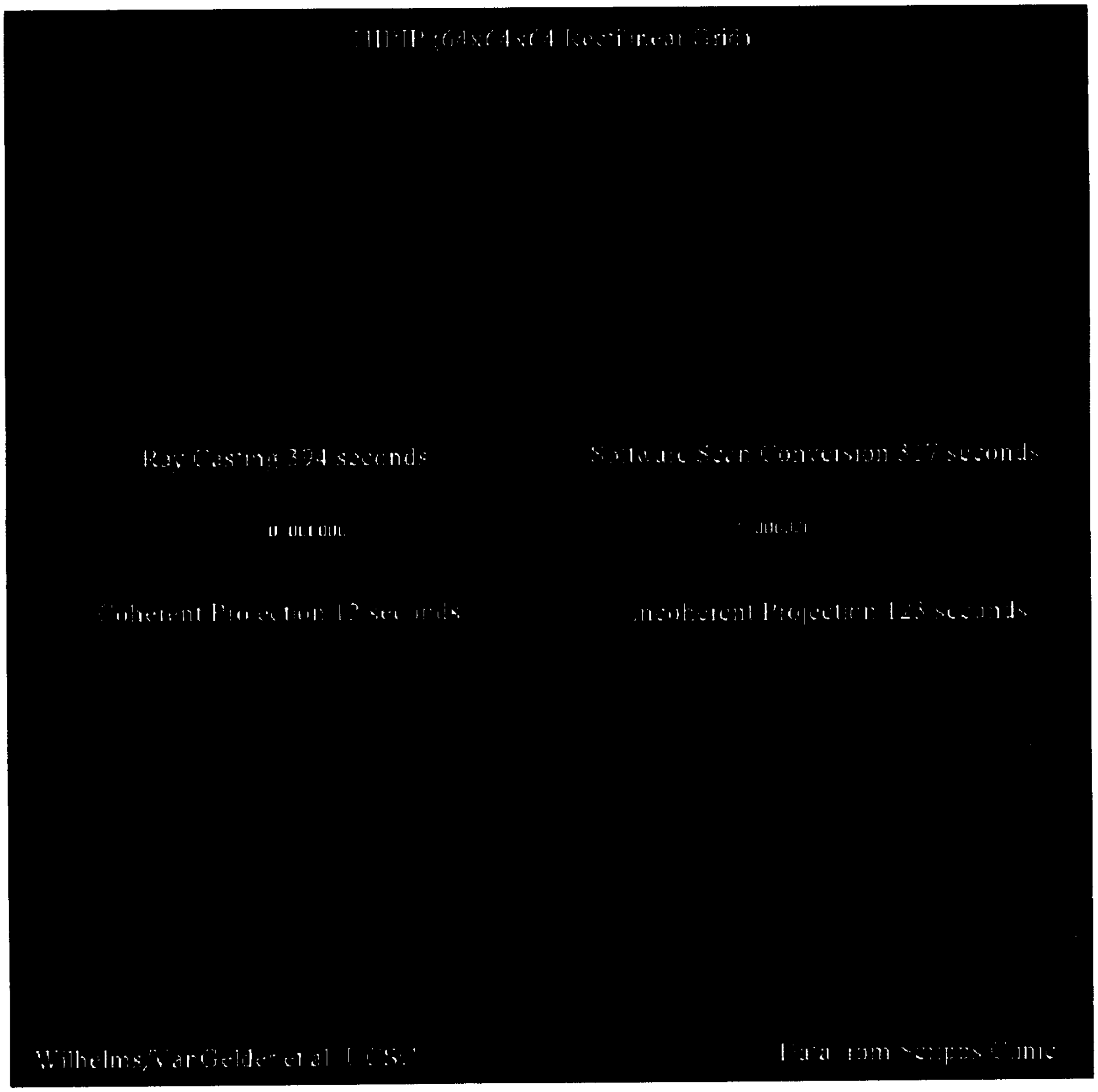
HIPIP (64x64x64 Rectilinear Grid)
Ray Casting 394 seconds
Software Scan Conversion 3.17 seconds
Coherent Projection 12 seconds
Incoherent Projection 128 seconds
Wilhelms/Van Gelder et al. UCSC
Data from Scripps Clinic

Figure 8: Comparison of four rendering methods on the rectilinear hipip volume.

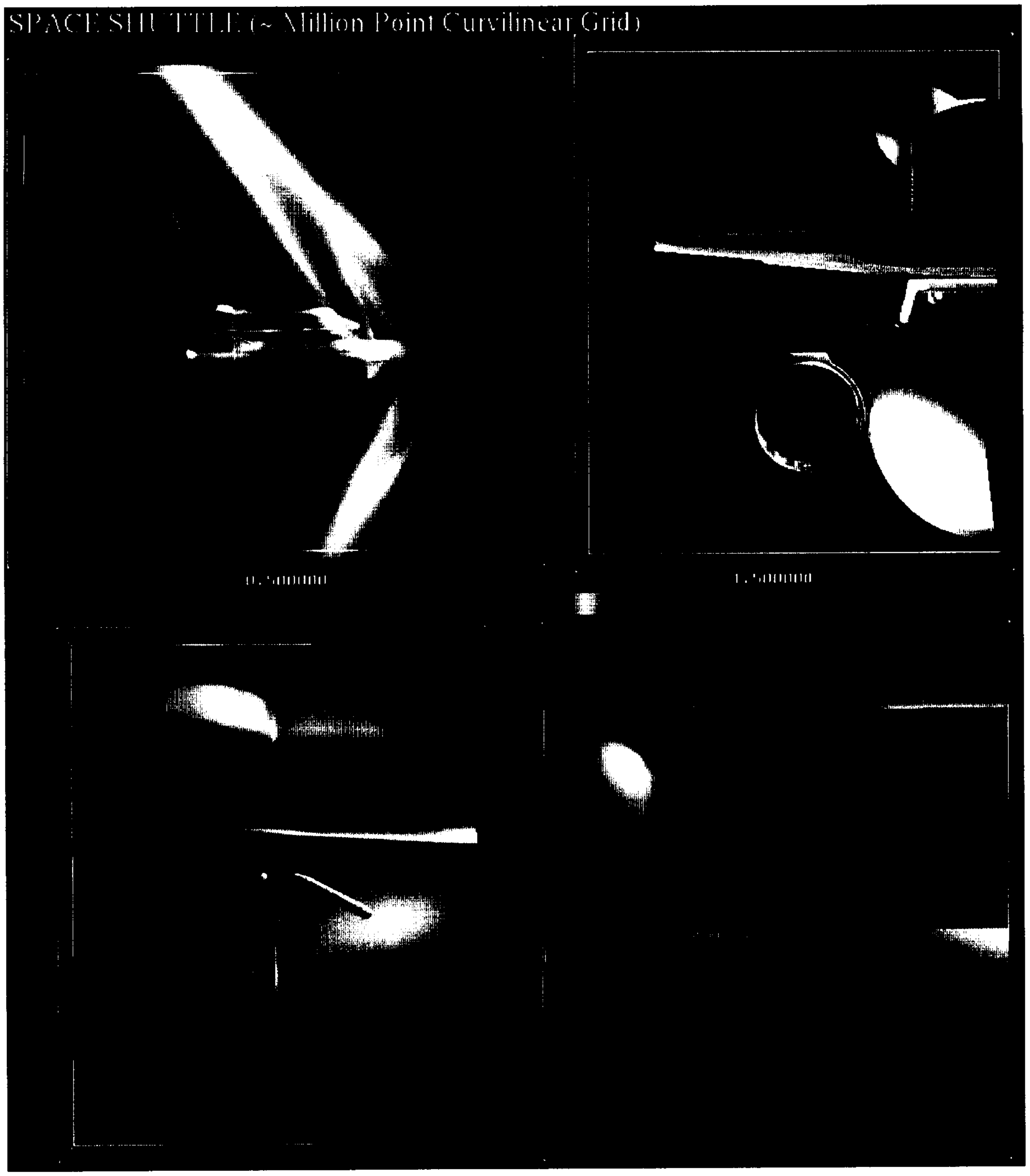

Figure 9: Four images of the space shuttle at different scales. Images took from 27 to 64 seconds on an SGI Onyx with four 150-MHz processors.

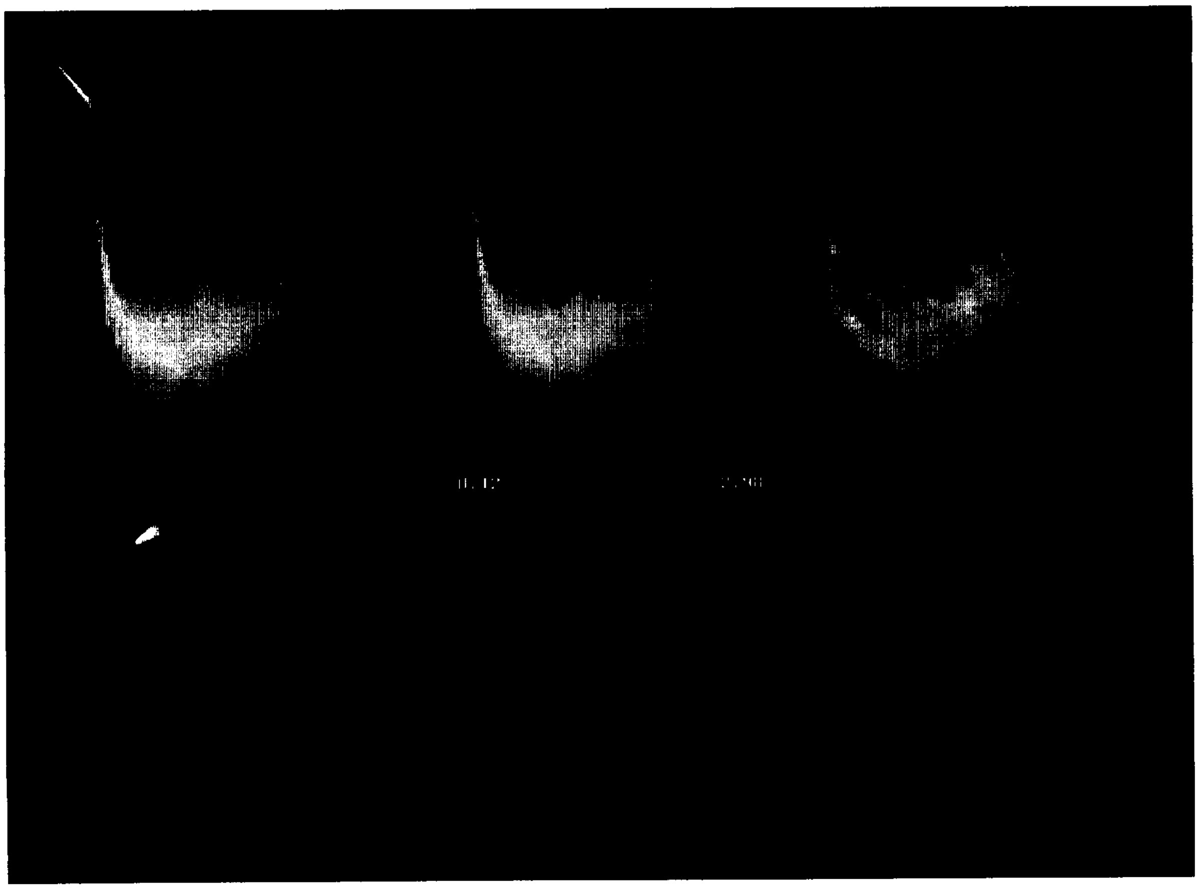

Figure 10: Software scan conversion of the fighter jet (a tetrahedral data set). The upper row shows the whole volume, and the lower row shows a close-up of an interesting region. The left column shows the volume and embedded surface; the center column shows the volume only. The right column is an approximated version (Section 3.4). While the left and center images took 38 seconds to render, the right images took less than one second, on an SGI Indigo-2 with a 200-MHz processor.